Objective Caml on .NET

The OCamIL Compiler and Toplevel

Raphaël Montelatici[1], Emmanuel Chailloux[2] and Bruno Pagano[3]

[1] Equipe PPS (UMR 7126) - Université Denis Diderot (Paris VII)

[2] Equipe PPS (UMR 7126) - Université Pierre et Marie Curie (Paris VI)

[3] Esterel Technologies

.NET Technologies'2005 - Plzen - May, 2005

Summary

- Motivations, Constraints and Choice
- Objective Caml
- OCamIL : Compiler and Toplevel
- O'Jacaré : Interoperability
- Related Work
- Current and Future Work

The OCamIL project

Motivations :

- To port O'Caml language and its libraries to the .NET platform

  and to be as compatible as possible with the standard (Inria implementation).

- To benefit from this platform : tools and interoperability

The OCamIL project

Contraints :

- To have a good level of compatibility. (compatibility)
- To produce managed code. (safety)
- Don’t modify the original language. (compatible)
- To follow perpetual evolutions. (easy evolutions)
- To have some facilities to interoperate. (interoperability)
- And if possible, to prevent inefficiencies. (efficiency)

The OCamIL project

Choice :

- OCamIL is developed as a new back-end of the O'Caml compiler.

Benefits are :

- More accurate compatibility
- Easier to write
- Easier to maintain

Drawbacks are :

- Sticks to O'Caml implementation choices
- Needs a retyping step
- Entails efficiency penalties

Objective Caml (O'Caml)

- One of the most popular ML dialect:
  - efficient code,
  - large set of general purpose and domain specific libraries,
  - automatic memory management,
  - used both for teaching (academy) and for writing high-tech applications (industry).
- Product of research results since 80Õs in: type theory, language design and implementation.
- Developped at INRIA (France). http://caml.inria.fr.

O'Caml features

- Functional language, exceptions, imperative extension.

O'Caml features

- Functional language, exceptions, imperative extension.
- High-level data types + pattern matching.

O'Caml features

- Functional language, exceptions, imperative extension.
- High-level data types + pattern matching.
- Polymorphism + *implicit* typing:
  - statically type-checked,
  - type inference,
  - polymorphic type (most general type is inferred).

O’Caml features

- Functional language, exceptions, imperative extension.
- High-level data types + pattern matching.
- Polymorphism + *implicit* typing:
  - statically type-checked,
  - type inference,
  - polymorphic type (most general type is inferred).
- OO extension.

O'Caml features

- Functional language, exceptions, imperative extension.
- High-level data types + pattern matching.
- Polymorphism + *implicit* typing:
  - statically type-checked,
  - type inference,
  - polymorphic type (most general type is inferred).
- OO extension.
- Multi-paradigm (inside the same typing mechanism):
  - Object-oriented (class structuration),
  - SML-like parametric module.

O'Caml : Examples of type inference

- functional type :

  ```
  let compose f g = fun x -> f (g x);;
  ```

  $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

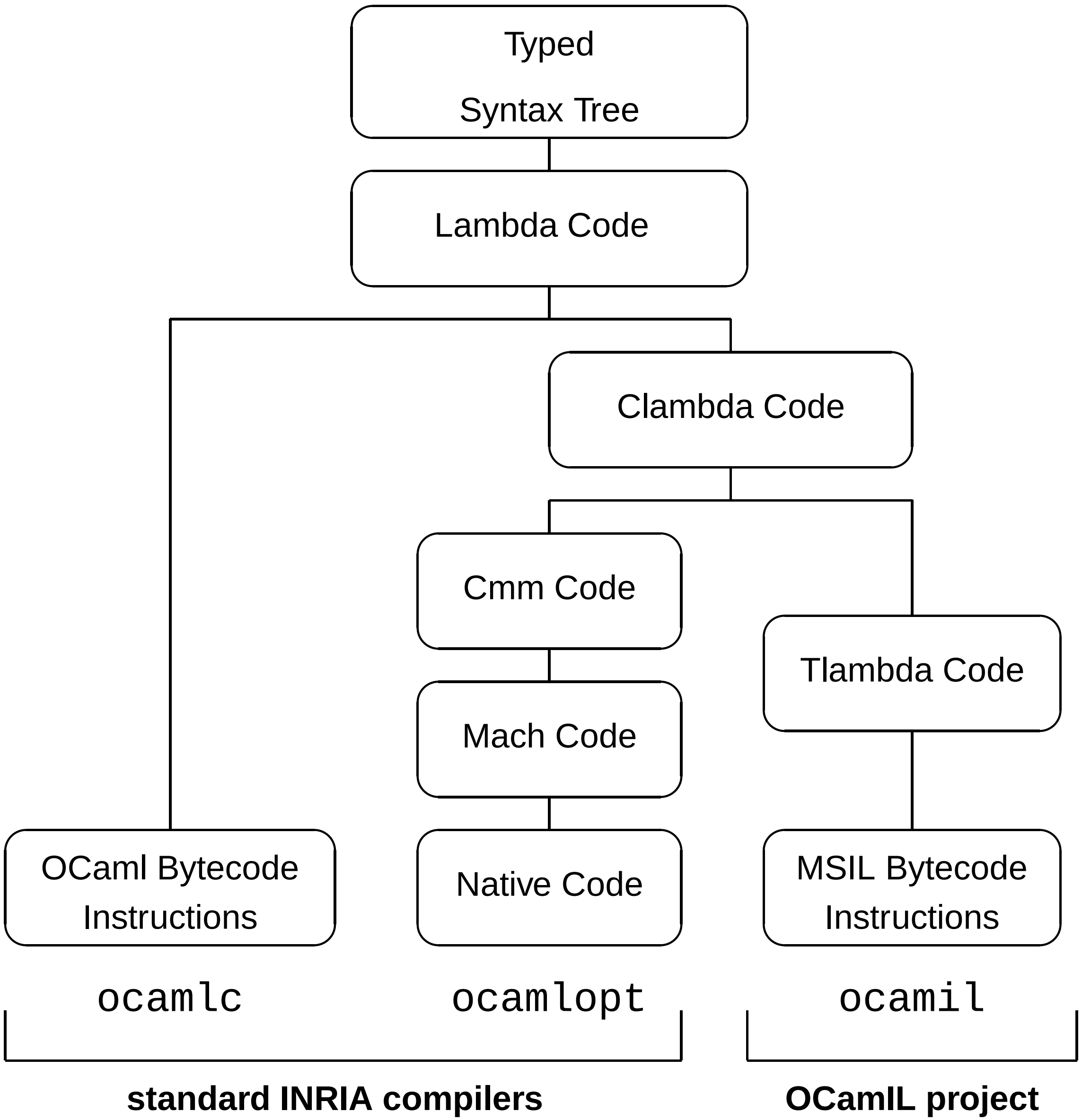
- functional type over list :

  `List.map` : $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`

- object and functional type :

  ```
  let toStringNL o = o#toString() ^"\n";;
  < toString : unit → string ; .. > → string
  let h = map toStringNL;;
  < toString : unit -> string; _.. > list  →
    string list
  ```

OCamIL: Architecture



OCamIL: CLambda

```
type ulambda =
    Uvar of Ident.t
  | Ulet of Ident.t * ulambda * ulambda
  | Udirect_apply of function_label * ulambda list
  | Ugeneric_apply of ulambda * ulambda list
  | Uclosure of
      (function_label * int * Ident.t list * ulambda) list
              * ulambda list
  | Uoffset of ulambda * int
  | Uprim of primitive * ulambda list
  | ...
```

λ-calcul : explicit closures management, control structures and primitives.

CLambda

```
type primitive =
    | Pmakeblock of int
    | Pfield of int
    | Psetfield of int
    | Psequand | Psequor | Pnot
    | Pnegint | Paddint | Psubint | Pmulint | Pdivint | Pmodint
    | Pandint | Porint | Pxorint | Plslint | Plsrint | Pasrint
    | Pintcomp of comparison
    | Pintoffloat | Pfloatofint | Pnegfloat | Pabsfloat
    | Paddfloat | Psubfloat | Pmulfloat | Pdivfloat
    | Pfloatcomp of comparison
    | Pstringlength | Pstringrefu | Pstringsetu
    | Pstringrefs | Pstringsets
    | ...
```

Typechecking

Main Problem :

- untyped intermediate language AND typed runtime
- Example : integers and blocks

Goals :

- To produce managed code without type erros
- To produce optimized code

Annotate primitives by types to ensure coherence.
2 approaches :

- to rebuild types
- OR to propagate types.

OCamIL V1 : type reconstruction

Minimal grammar for types :

```
T ::= int | block | string
    | float | closure | unit | any
```

- To discriminate kinds of blocs
- A different representation (than O'Caml) for sum types.

Implementation :

O'Caml	bool/int	float	string	unit	->	'a	*others*
retyping	int	float	string	unit	closure	any	block
CTS	int32	float64	StringB	null/void	Closure	object	object[]

OCamIL V1 : type reconstruction

For sum types, we want an union type :

$$\texttt{sumtype} = \texttt{int} \cup \texttt{block}$$

Problem :

```
type t = Zero | One | Node of t
```

O'Caml code	Clambda code

`let cut = function`
`  | Node n -> n`
`  | x -> x`
	`let cut = closure(cut):`
	`     x -> if (isint x) then x`
	`          else (field 0 x)`

Type: `t -> t`	Inferred type: `sumtype -> sumtype`

OCamIL V1 : type reconstruction

```
type t = Zero | One | Node of t
```

O'Caml code	Clambda code

`let hell a b =`
`  match a with`
`    | Zero -> One`
`    | _ -> b`
	`let hell = closure(hell): a -> b ->`
	`     if (isint a) then`
	`        (if (a != 0) then b else 1)`
	`     else b`

Type: `t -> t -> t`	Inferred type: `sumtype -> int -> int`

the following expression produces an incoherence:

```
hell One (Node Zero)
```

OCamIL V1 : type reconstruction

Solution : to modify sum types representation.

- To represent all constructors (including constants) by blocs.
- More elegant solutions don’t work (because polymorphic variants)
- Une solution plus fine ne marcherait pas (à cause des variants polymorphes)

It is a first modification before our back-end.

Other Main Implementation Choices

Uniform Representation:

- All values can be represented by an `Object`
- boxing/unboxing for scalar values

Closures:
- Each abstraction (`fun`) becomes a class,
- sub-class of the abstract class `Closure`.
- A closure is an intance of their corresponding class.

Application: :

- direct apply
- using a general mechanism :
  - which can create a new closure (with a richer environment)
  - or can execute the real code (body of abstraction)

Other Main Implementation Choices

Exceptions:

- sub-classes of .NET exceptions

Objects:

- only records (instance variables and virtual methods table)
- general application :

  `o.m a1 a2 ==> GET(o,m) o a1 a2`

Functors:

- represented by closures too

Maturity

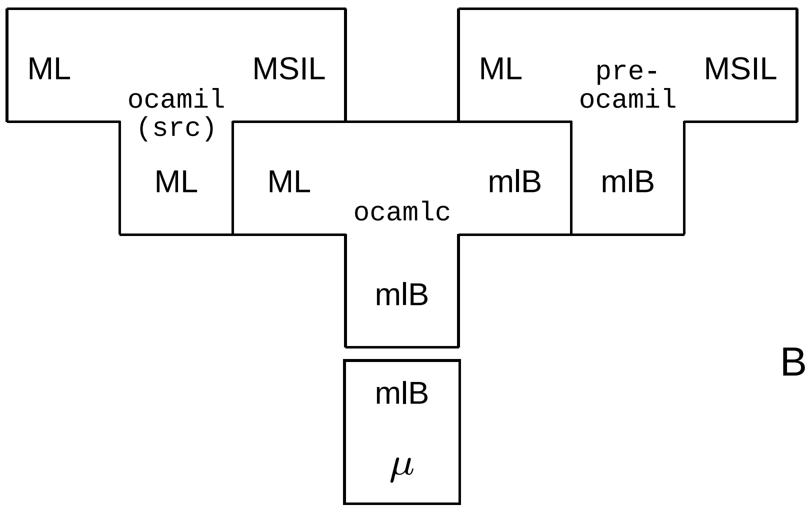
Libraries :

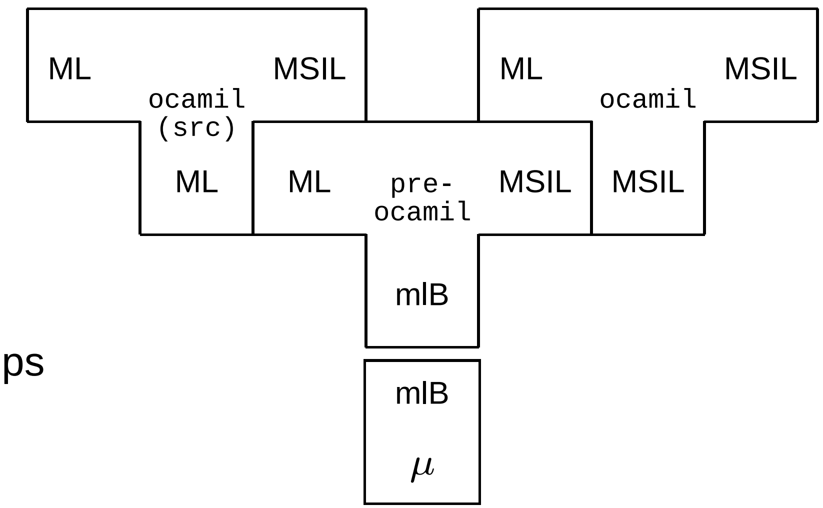
- standard library
- Graphics
- Threads
- Dynlink

Compatibility :

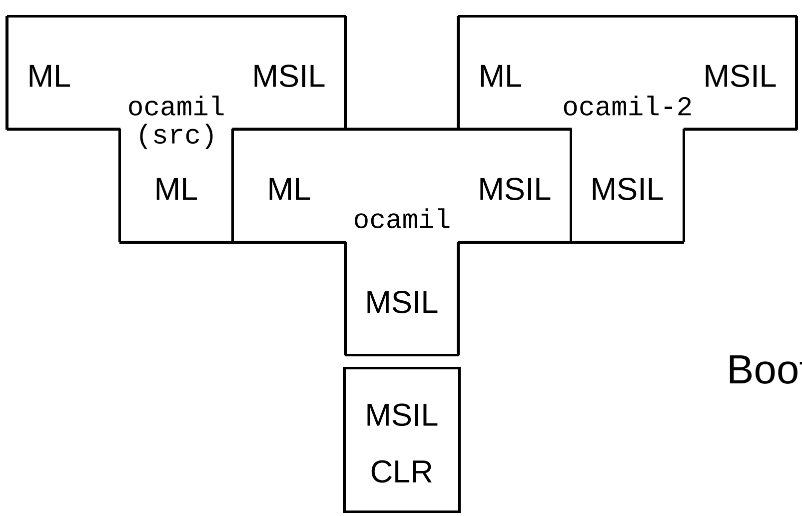
- Bootstrapped compiler
- Toplevel

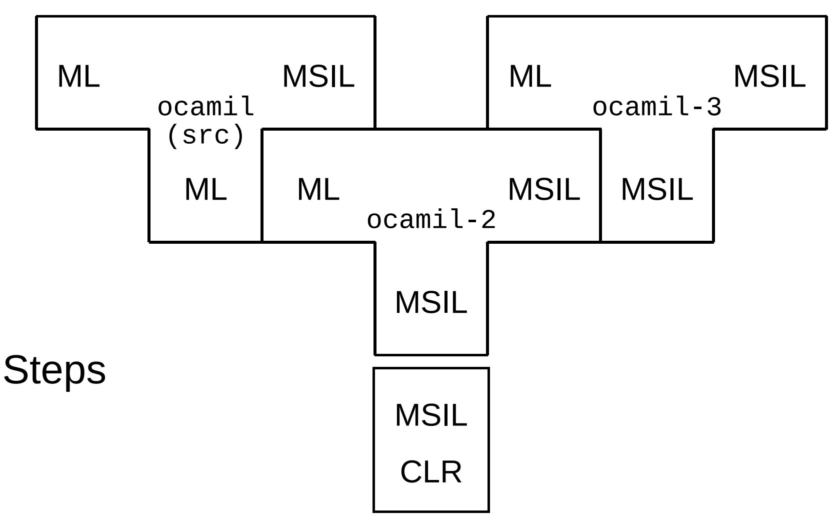
Bootstrap





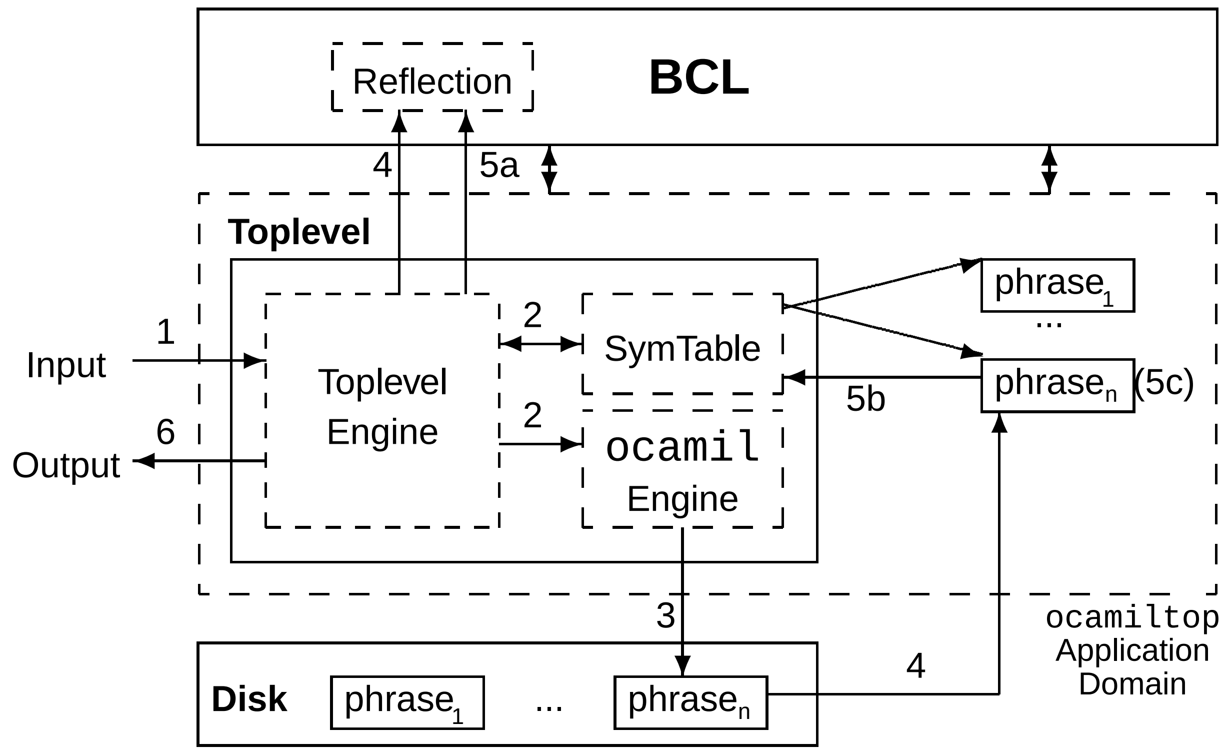
Building Steps





Bootstrapping Steps

Toplevel



Reflection
BCL
4
5a
Toplevel
1
Input
6
Output
Toplevel
Engine
2
SymTable
2
ocamil
Engine
phrase1
...
phrasen
(5c)
5b
3
ocamiltop
Application
Domain
Disk
phrase1
...
phrasen
4

Interoperability

low-level FFI, using static methods .NET.

```
external il_getenv: string -> string =
"string" "System.Environment::GetEnvironmentVariable"
         "string"
;;

let getenv var = match il_getenv var with
    | "" -> raise Not_found
    | s -> s
;;
```

Example : implementation of `Sys.getenv`

Interoperability : Low Level FFI

```
(BSCAMIL) Objective Caml version 3.06+camil
# let zodiac = List.map (fun i -> String.make 1 (char_of_int i))
[0x9f20;0x725b;0x864e;0x5154;0x9f99;0x86c7;0x9a6c;0x7f8a;0x7334;0x9e21;0x72d7;0x732a];;
val zodiac : string list = ["鼠"; "牛"; "虎"; "兔"; "龙"; "蛇"; "马"; "羊"; "猴"; "鸡"; "狗"; "猪"]
# List.sort String.compare zodiac;;
- : String.t list =["兔"; "牛"; "狗"; "猪"; "猴"; "羊"; "虎"; "蛇"; "马"; "鸡"; "鼠"; "龙"]
# type culture_info;;
type culture_info
# external create_culture:string -> culture_info = "class System.Globalization.CultureInfo"
"System.Globalization.CultureInfo" "CreateSpecificCulture" "string";;
external create_culture : string -> culture_info = "CreateSpecificCulture" "CreateSpecificCulture"
# external uni_compare:string -> string -> bool -> culture_info -> int = "int" "System.String"
"Compare" "string" "string" "bool" "class System.Globalization.CultureInfo";;
external uni_compare : string -> string -> bool -> culture_info -> int = "Compare" "Compare"
# let pinyin_compare s1 s2 =
      let chinese = create_culture "zh-CN" in
          uni_compare s1 s2 true chinese;;
val pinyin_compare : string -> string -> int = <fun>
# List.sort pinyin_compare zodiac;;
- : String.t list =["狗"; "猴"; "虎"; "鸡"; "龙"; "马"; "牛"; "蛇"; "鼠"; "兔"; "羊"; "猪"]
#
```

Interoperability

Features	C#	O'Caml	Features	C#	O'Caml
classes	$\surd$	$\surd$	inheritance $\equiv$ sub-typing?	yes	no
late binding	$\surd$	$\surd$	overloading	$\surd$	*
early binding	$\surd$	*	multiple inheritance	*	$\surd$
static typing	$\surd$	$\surd$	parametric classes	*	$\surd$
dynamic typing	$\surd$	*	packages/modules	*	*
sub-typing	$\surd$	$\surd$			

IDL : an intersection of the two models : for which inheritance and subtypingrelations are equivalent, overloading and binary methods are not allowed. No multiple inheritance nor parametric classes.

O'Jacaré.Net : Class Point

File `point.idl`

```
class Point {
    int x;

    [name default_point] <init> ();
    [name point] <init> (int);

    void moveTo(int);
    string toString();
    boolean eq(Point);
}
```

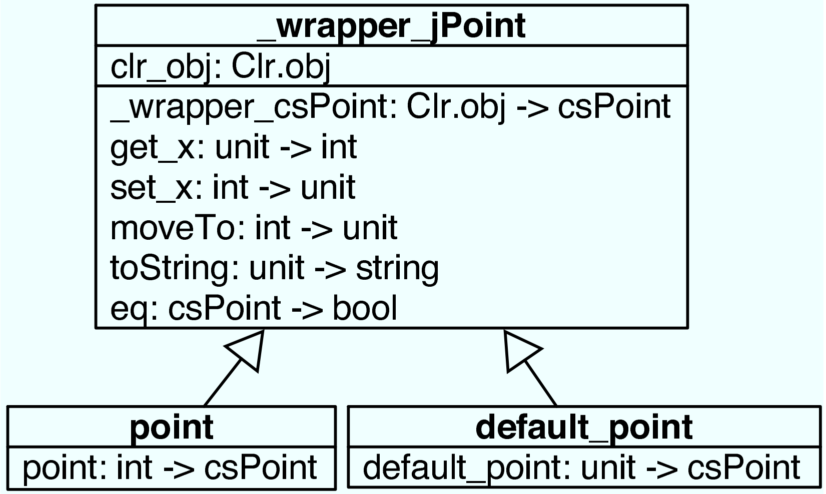
Generates : `point.ml`

Object type `csPoint`

Wrapper `_wrapper_csPoint`

Users classes

`default_point`, `point`



O’Jacaré.Net : Class ColoredPoint

File `point.idl`	Generates : `point.ml`

class ColoredPoint **extends** Point

implements Colored {

[**name** default_colored_point]

<**init**> ();

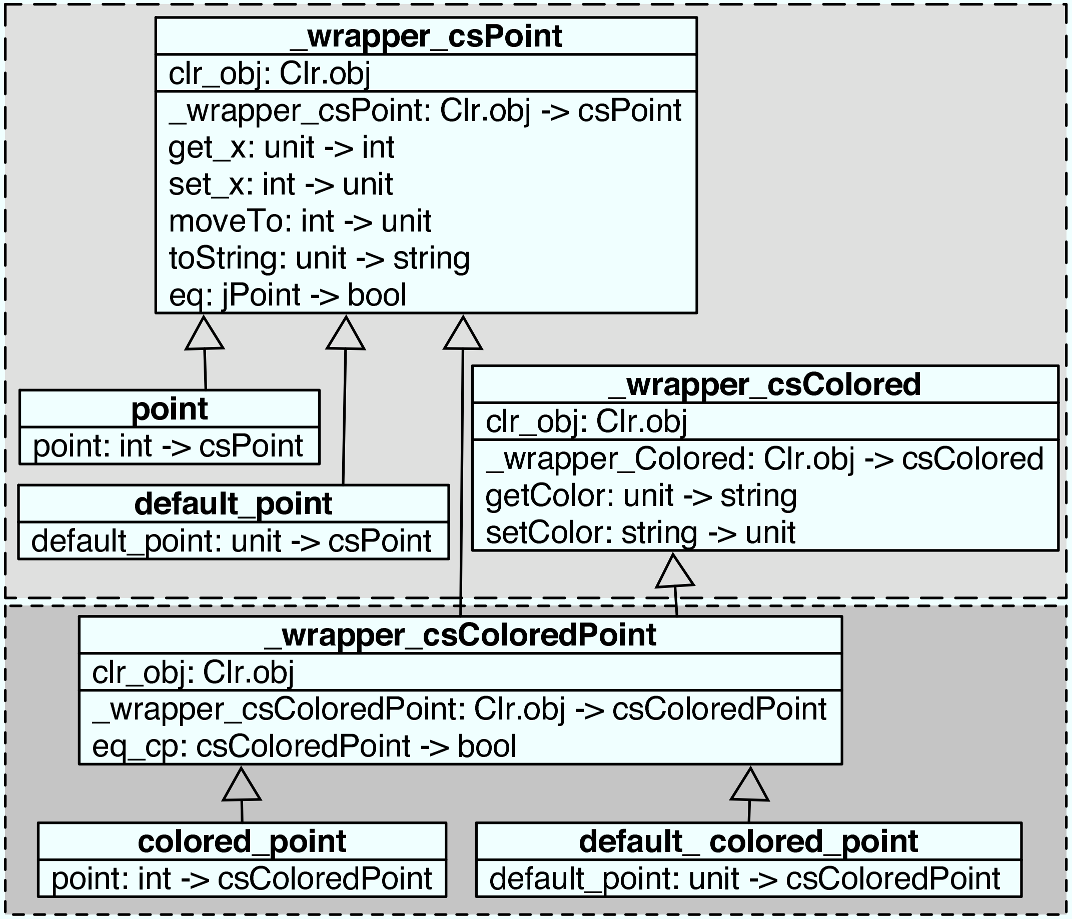
[**name** colored_point]

<**init**> (**int**,**string**);

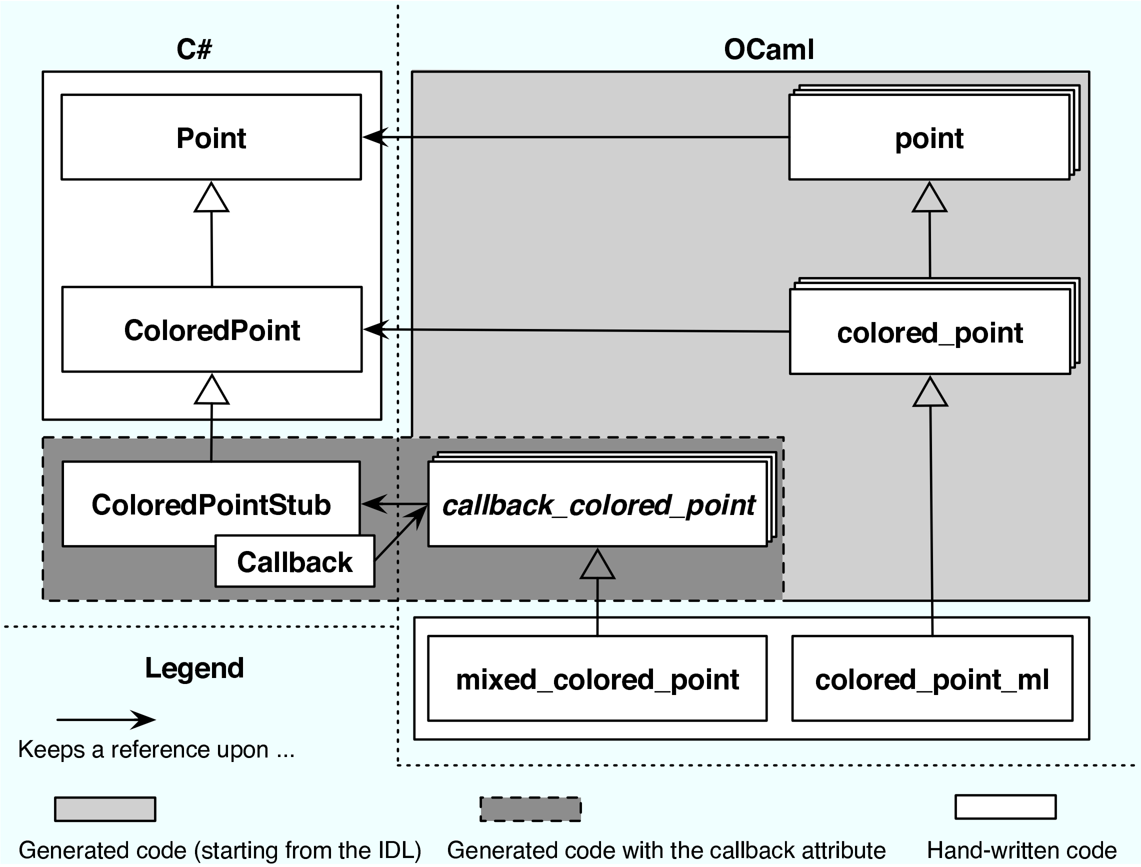
[**name** eq_cp]

boolean eq(ColoredPoint);

}



Interoperability



Combining the two Objects Models

- Multiple inheritance of C# classes.
- Downcasting C# objects in O'Caml.

Multiple inheritance of C# classes

The file `rect.idl`	The O'Caml program

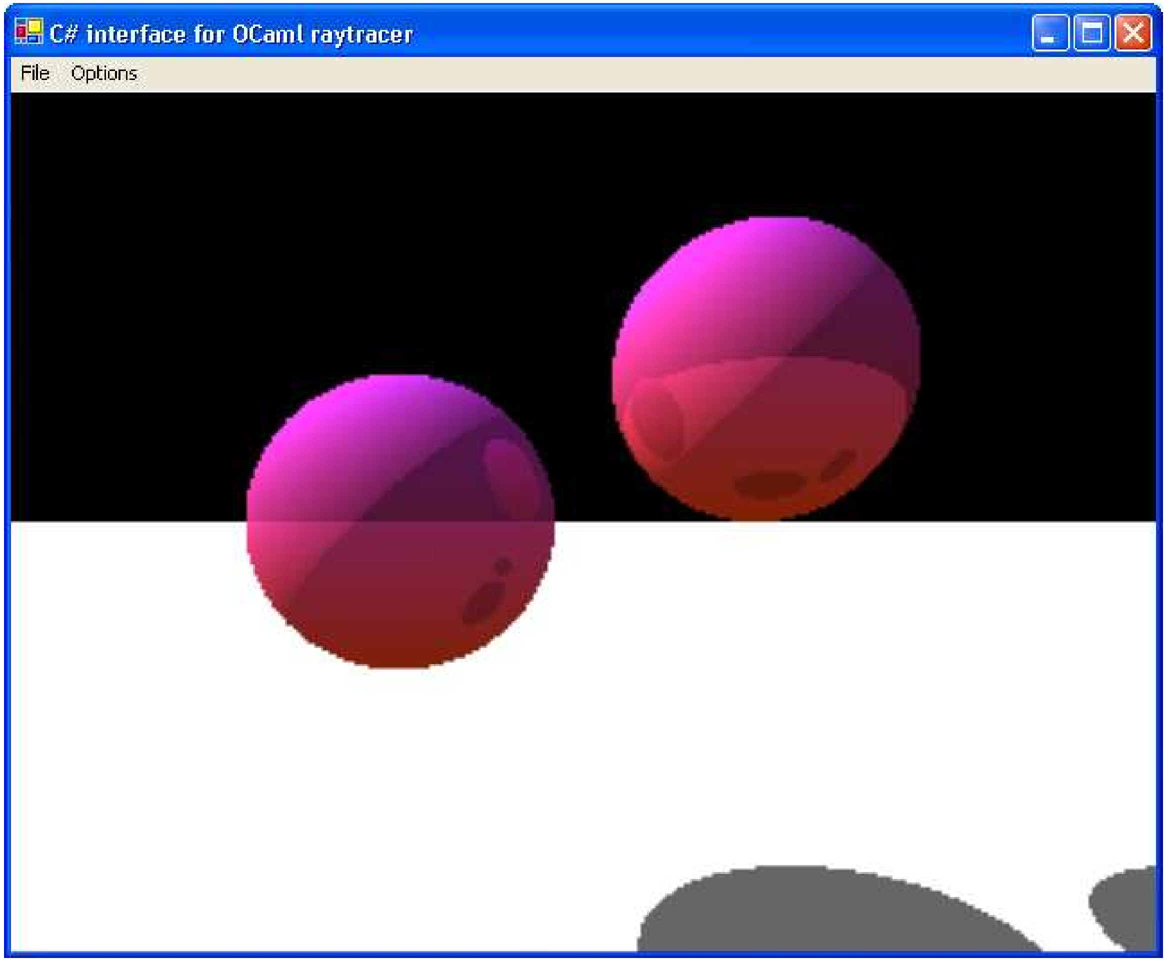
<pre>**package** mypack;
 	
class Point {
  [**name** point] <**init**> (**int**, **int**);
}
class GraphRectangle {
  [**name** graph_rect] <**init**>(Point, Point);
  string toString();
}
class GeomRectangle {
  [**name** geom_rect] <**init**>(Point, Point);
  double area();
}</pre>
	<pre>**open** Rect;;
	class geom_graph_rect p1 p2 =
	object
	  inherit geom_rect p1 p2 **as** super_geo
	  inherit graph_rect p1 p2 **as** super_graph
	end;;
	
	let p1 = **new** point 10 10;;
	let p2 = **new** point 20 20;;
	let ggr = **new** geom_graph_rect p1 p2;;
	Printf.printf "area=%g\n" (ggr#area ());;
	Printf.printf "toString=%s\n" (ggr#toString ());;</pre>

Downcasting C# objects in O'Caml

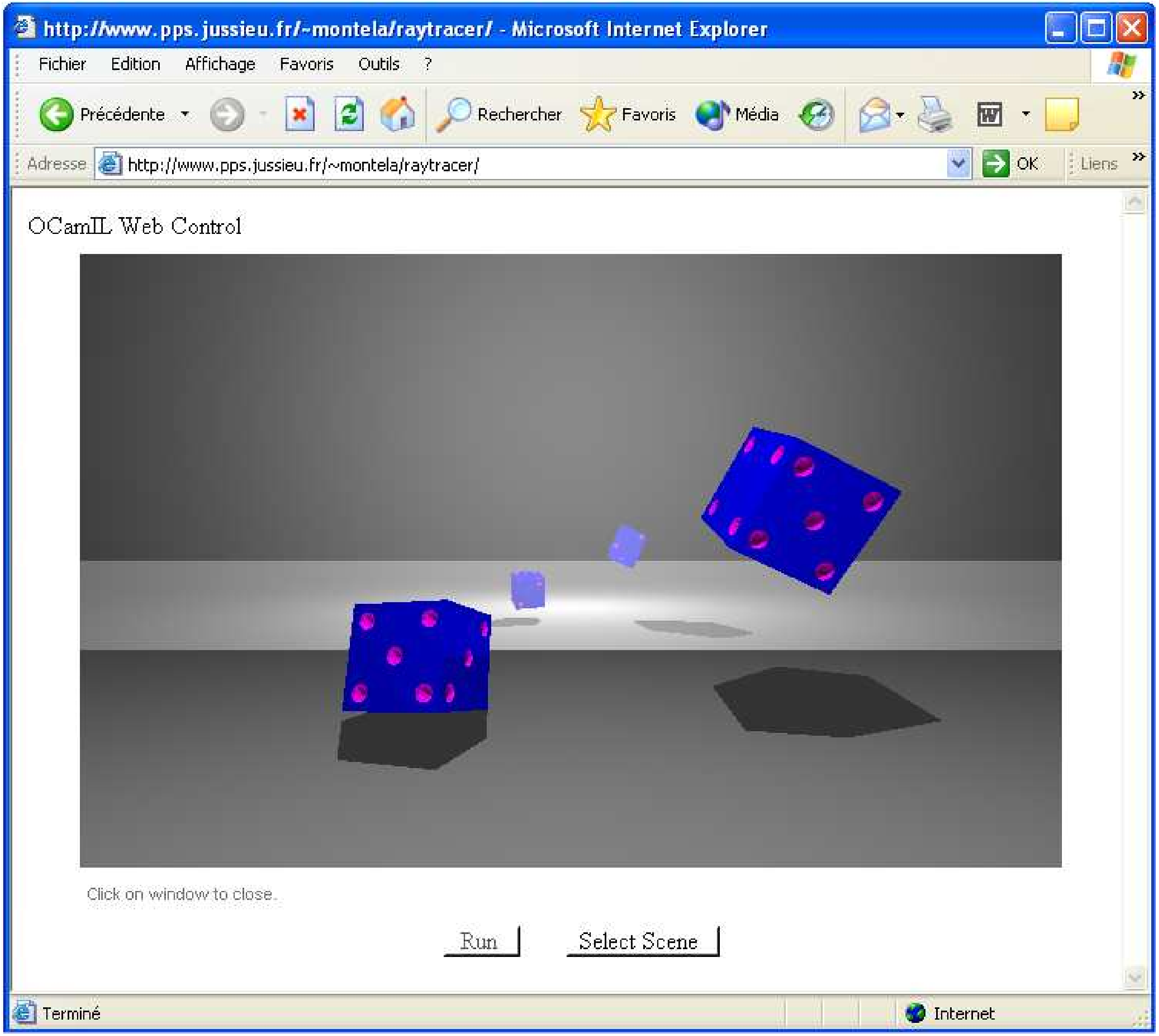
```
let l = [(ml_cp :> csPoint); (wml_cp :> csPoint)];;
val l : csPoint list = <obj>
let lc = List.map (fun x -> csColoredPoint_of_top (x :> top)) l;;
val l : csColoredPoint list = <obj>
```

- The generated O'Caml class hierarchy has root class `top`,
- O'Jacaré.Net defines type coercion functions from `top` to child classes.

Interoperability



Interoperability



Related Work

- F#
  - An Caml-Light Core : no functors, no O'Caml objects.
  - Nombreuses autres petites incompatibilités
  - Well integrated C# object model (not the O'Caml model)
  - no toplevel
  - similar efficiency
- SML.NET
  - compatibility with SML
  - integration of a C# syntax
  - no toplevel
  - better efficiency (global monomorphisation)
  - SML $\neq$ O'Caml ...

Current Work

to propagate types from types syntax tree :

- introducing a typed intermediate language

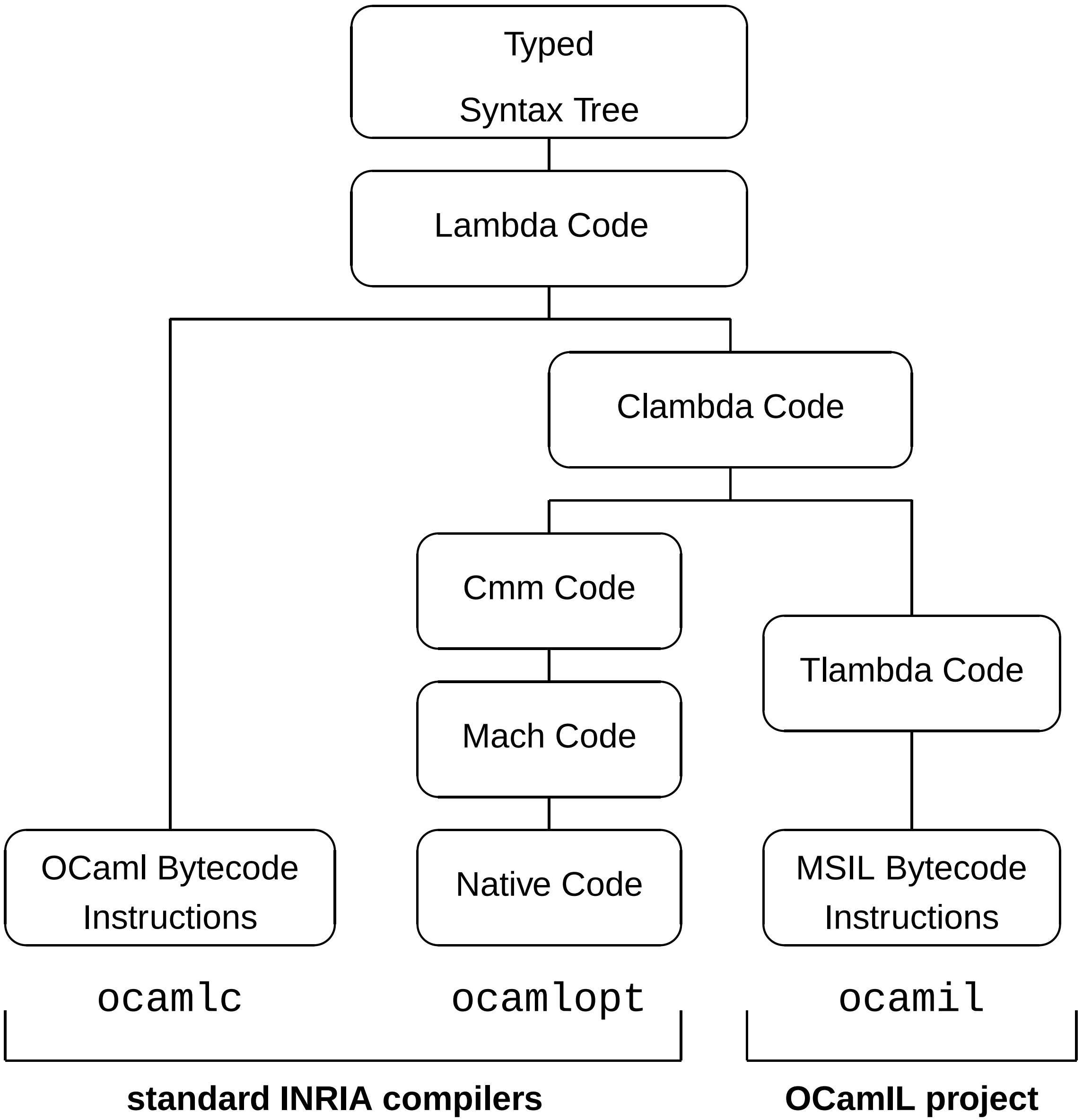
interest :

- allows to represent more precisely O'Caml types
- better efficiency and easier debug

difficultgy :

- propagation during all optimazation steps (pattern matching, . . .)

OCamIL V2 : type propagation



OCamIL V2 : type propagation

Type information is available in syntax tree, but is not propagating in intermediate languages.

```
type t = A of int * t | B of int

let f = function
    A(0,u) , B i -> A(i,u)
  | ( A(x,_) , A(y,_) | B x , B y ) -> B(x+y)
  | A(_,u) , _ -> u
  | _ , _ -> B 0
```

OCamIL V2 : type propagation

```
param ->
  catch 1 x' y'
    let left = (field 0 param) in
       switch left
          0 -> let x = (field 0 left) in
                  catch 3
                    if (x != 0) then <jump 3> else
                       let right0 = (field 1 param) in
                          switch right0
                            0 -> <jump 3>
                            1 -> [makeblock 0 (field 0 right0)
                                                 (field 1 left)]
                  c-with(3) let right1 = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (field 0 right1)>
                                 1 -> (field 1 left)
          1 -> let right = (field 1 param) in
                  switch right
                     0 -> [1: 0]
                     1 -> <jump 1 (field 0 left) (field 0 right)>
  c-with(1) [makeblock 1 (x'+y')]
```

OCamIL V2 : type propagation

```
param:t*t ->
  catch 1 (x':int) (y':int)
    let left:t = (field 0 param) in
       switch left
          0 -> let x:int = (field 0 left) in
                  catch 3
                    if (x != (0:int)):bool then <jump 3>:t else
                       let right0:t = (field 1 param) in
                          switch right0
                            0 -> <jump 3>:t
                            1 -> [makeblock 0 (field 0 right0):int
                                              (field 1 left):t]:t
                  c-with(3) let right1:t = (field 1 param) in
                                 switch right1
                                   0 -> <jump 1 x (field 0 right1):int>:t
                                   1 -> (field 1 left):t
          1 -> let right:t = (field 1 param) in
                  switch right
                     0 -> [1: 0]:t
                     1 -> <jump 1 (field 0 left):int (field 0 right):int>:t
  c-with(1) [makeblock 1 (x'+y'):int]:t
```

OCamIL V2 : type propagation

```
param:t*t ->
  catch 1 (x':int) (y':int)
    let left:t = (field 0 param) in
      switch left
        0 -> let x:int = (field 0 left:t.A) in
                catch 3
                  if (x != (0:int)):bool then <jump 3>:t else
                     let right0:t = (field 1 param) in
                        switch right0
                          0 -> <jump 3>:t
                          1 -> [makeblock 0 (field 0 right0:t.B):int
                                            (field 1 left:t.A):t]:t
                c-with(3) let right1:t = (field 1 param) in
                               switch right1
                                 0 -> <jump 1 x (field 0 right1:t.A):int>:t
                                 1 -> (field 1 left:t.A):t
        1 -> let right:t = (field 1 param) in
                switch right
                   0 -> [1: 0]:t
                   1 -> <jump 1 (field 0 left:t.A):int (field 0 right:t.B)
  c-with(1) [makeblock 1 (x'+y'):int]:t
```

OCamIL V2 : type propagation

Grammar for retyping :

O'Caml	`type 'a r = {x:'a; y:int; z:('a*'a) r}`
retypage	`record(r, x:any, y:int, z:r)`
CTS	`class r {object x; int y; r z;}`

O'Caml	`type t = A of t * t | B of int`

retypage	`sumtype(t, t.A, t.B)`
	`record(t.A, x0:t, x1:t)`
	`record(t.B, x0:int)`

CTS	`class t {int get_tag();}`
	`class t_A inherits t {t x0; t x1;}`
	`class t_B inherits t {int x0;}`

OCamIL V2 : type propagation

- More precise types for primitives and best representation for sum types and records.
- More difficult to implement
- deeper modifications of the O'Caml compiler

Benchmarks

The following benchmarks ran on a Windows XP Pentium IV 2,4GHz station. They are designed to run in about a second under the native O'Caml compiler (`ocamlopt`).

	ocamlopt	ocamlc	OCamlILR|	OCamlILP	F#	sml.net
Boyer	1.45	5.21	55.8	50.4	52.1	36.8
KBGr	1.14	1.31	11.9	11.7	11.8	6.18
KBGeo	1,34	2.75	59.3	59.2	66.1	28.5
Nucleic	0.81	5.82	11.5	7.62	7.48	0.81

Performance tests (real time in seconds).

Future Work

- To complete the types propagation version, and to compare both
- To implement some optimizations (closures, exceptions) and to use generic IL.
- to build a debugger thaht explores O'Caml values
- to parameterize backend to produce other byte-codes